A fault tolerant tightly coupled multiprocessor architecture based on stable transactional memory

Michel Banâtre, Philippe Joubert

INRIA

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports de Recherche

N° 1178

Programme 3
Réseaux et Systèmes Répartis

A FAULT TOLERANT TIGHTLY COUPLED MULTIPROCESSOR ARCHITECTURE BASED ON STABLE TRANSACTIONAL MEMORY

Michel BANATRE
Philippe JOUBERT

Mars 1990

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042-RENNES CÉDEX
FRANCE
Téléphone: 99 36 20 00
Télex: UNIRISA 950 473 F
Télécopie: 99 38 38 32

Publication Interne n° 512
Février 1990 - 20 Pages

A Fault Tolerant Tightly Coupled Multiprocessor Architecture based on Stable Transactional Memory

Une architecture multiprocesseur fortement couplée tolérante aux fautes fondée sur l'utilisation d'une mémoire transactionnelle stable

Michel Banâtre IRISA-INRIA Rennes
Philippe Joubert IRISA Rennes

Campus universitaire de Beaulieu,
35042 Rennes Cedex (France)

Abstract

Traditionally, tightly coupled multiprocessors allow data sharing between multiple caches by keeping cached copies of memory blocks coherent with respect to shared memory. This is difficult to achieve in a fault tolerant environment due to the need to save global checkpoints in shared memory from where consistent cache states can be recovered after a failure. The architecture presented in this report solves this problem by encapsulating the memory modifications done by a process into an atomic transaction. Caches record dependencies between the transactions associated with processes modifying the same memory blocks. Dependent transactions may then be atomically committed. Such an operation requires a cache coherence protocol responsible for recording process dependencies as well as keeping coherent cached copies of blocks and a shared *Stable Transactional Memory* owing to which all memory updates are atomic to allow recovery after a processor failure.

This report also presents potential performance for this architecture the data for which was obtained through simulation.

Key words

Tightly coupled multiprocessors, fault tolerance, cache coherence protocols, stable transactional memory, atomic cache flush.

.../...

Résumé

L'un des principaux intérets des multiprocesseurs fortement couplés est de permettre à plusieurs caches de partager efficacement des données tout en assurant la cohérence de ces données. Cette propriété est difficile à assurer dans une architecture fortement couplée tolérant les fautes du fait de la nécessité de sauvegarder en mémoire partagée des points de reprise globaux qui sont nécessaires pour reconstruire un état cohérent des caches après une panne. L'architecture présentée dans ce rapport résoud ce problème en encapsulant les modifications mémoire effectuées par un processus dans une transaction atomique. Les caches enregistrent des dépendances entre les transactions associées à des processus modifiant les mêmes blocs. Les transactions dépendantes peuvent alors être validées de façon atomique grâce à une *mémoire transactionnelle stable.*

Ce rapport présente aussi une estimation par simulation des performances de cette architecture.

Mots clés

Multiprocesseurs fortement couplés, tolérance aux fautes, protocoles de cohérence de caches, mémoire transactionnelle stable, vidage atomique de caches.

1 Introduction

This report presents the architecture of a fault tolerant shared memory multiprocessor, the tc_FTMP (tightly coupled Fault Tolerant Multiprocessor). In this architecture, process states are locally modified within non write through caches and atomically updated into a stable memory by means to an hardware transaction mechanism. Sharing of memory blocks between multiple processes is handled by a cache coherence protocol very close to those used in standard shared memory multiprocessors. Caches log dependencies between processes sharing memory blocks and dependent process states are updated atomically. Though this architecture is not currently implemented, we have developed a simulation model allowing us to compare its predictable performance with those of other multiprocessor architectures.

This report is organized as follows: section 2 reviews background and previous work. Section 3 presents the architecture and operation of tc_FTMP and of its associated caches and stable transactional memory. Section 4 discusses potential performance for this architecture, the data for which was obtained through simulation.

2 Background

In order to provide powerful and reliable computers, many fault tolerant multiprocessor architectures have been developed in recent years. Multiprocessor architectures can be divided into two categories : (i) *loosely coupled*, where each processor has private resources (memory, I/O devices etc.) and communicates with other processors through message passing protocols; and (ii) *tightly coupled*, where each processor directly accesses all memory and I/O resources and communicates with other processors through shared memory. Apart from n-modular redundancy techniques like the one proposed in the System/88 [Harr87] or FTMP [Hopk78] computer which are valid for all types of multiprocessor, different means for tolerating processor failures are adopted dependent upon the type of architecture.

One method of achieving fault tolerance on loosely coupled architectures uses a process-pair scheme. Every process running in the system is backed up on another processor. A primary process executes the main computation and is periodically synchronized with an inactive backup process that holds checkpoints which are process states from which computation may be safely restarted if the processor running the primary process fails. Tandem [Bart87] and the recent Targon/32 system [Borg89] use loosely coupled architecture and process-pairs.

Another method is to hold process checkpoints into a stable storage device [Bana88]. Stable storage [Lamp76] is a memory device whose physical storage is stable (ie., information does not degrade over time) and whose write operation is atomic. In this scheme, each processor is equipped with a stable memory board into which it periodically stores process checkpoints. No backup processes are needed since the stable memory associated with each processor can be accessed by another processor through a global bus to retrieve checkpoints after a processor crash.

To our knowledge, there was only one previously proposed scheme for fault tolerant

tightly coupled multiprocessors not using n-modular redundancy techniques : the Sequoia architecture [Bern88]. In this architecture, each processor is associated with a non write through cache memory [Smit82] allowing multiple writes on a memory block before main memory is updated. Each processor performs memory updates locally within its cache and periodically checkpoints its state by flushing the cache and its internal registers to main memory.

Modified data is flushed on two distinct memory modules to handle memory failures and processor failures during a flush operation. If a processor fails, another processor resumes the work of the failed one by using its last checkpoint saved in shared memory. Multiple processors modify shared memory locations through explicit locking of hardware Test-And-Set locks associated with each memory module. The protocol requires the following steps :

1. Get lock on memory location to be modified (with a possible busy wait loop until lock is free).
2. Invalidate non dirty cache.
3. Perform critical section code.
4. Flush (dirty) cache.
5. Release lock.

Because after a flush, cached data remains valid for future access (ie, is not invalidated), steps (2) and (4) are necessary to ensure that a processor reads the last version of a block. Lockable resources are partioned to reduce lock contention.

This protocol requires a heavy overhead to access shared memory locations and thus limits the use of the traditional advantage of tightly coupled multiprocessors : easy data sharing and memory coherence, generally achieved through *cache coherence protocols* [Arch87].

Our goal is to build a fault tolerant tightly coupled multiprocessor which is able to take advantage of the data sharing facilities provided by non fault tolerant shared memory multiprocessors with the help of a stable memory device supporting atomic transactions.

Previous researches have lead us to study and implement stable memory devices. The first device was for the ENCHERE electronic marketing system [Bana86] and consisted of dual random access memory banks updated atomically. Next we built a more sophisticated version [Bana88] which permits atomic operations on objects (record, arrays, etc) as well as on groups of objects. This stable storage survives processor hardware and software faults by rejecting incorrect memory accesses. In order to provide atomicity of updates, every object possesses two copies located on two physically independent memory banks. The processor updates an object by writing into the first memory bank. If no errors are detected, the stable memory copies atomically the object from the first bank onto the second bank. If the processor fails during the update, a consistent version of the object may be retrieved from either the first or the second bank.

The next step is a extension of this stable memory allowing, several atomic transactions on groups of objects to run concurrently. This stable memory is used by the tc_FTMP architecture.

3 The tc_FTMP architecture

3.1 Overview

To achieve fault tolerance, the tc_FTMP architecture keeps main memory state coherent at all times by encapsulating memory updates into *atomic transactions* implemented using a shared *stable transactional memory* (STM). The STM is the main memory of the architecture.

The atomic transaction - or atomic action - mechanism provided by the STM is very close to those described in [Lamp81] and [Rand78], but at cache and operating system level. From the STM point of view, a transaction consists of the atomic modification of a group of memory blocks in the STM. The STM guarantees the atomicity of this modification. This implies the ability for the STM to identify the blocks to be atomically updated within each transaction.

To construct consistent checkpoints for process (that is, memory states from which failed processes may be safely restarted), its is necessary to atomically update into the STM all the blocks accessed by a process during a given period of time.

The basic idea is to associate at least one transaction T_P with each process P at any time. The memory blocks modified by P are kept within the cache associated with each processor and are from time to time atomically written back into the STM, establishing the checkpoint required to restart P in case of a failure of the processor running P. The STM associates a time out with each transaction. A failure is detected by the STM when a transaction is timed out. In the following, a block is said to be *part* of a transaction if it has been modified by the process associated with that transaction.

To ensure that all the blocks modified by P are written into the STM, the STM records all write accesses made on blocks by P within T_P and atomically updates those blocks when T_P commits.

Transaction commit generally occurs when the cache is full of modified data. To make room for new blocks it is necessary to write back some blocks to main memory. At this point, the cache chooses one of the processes it runs, writes all the blocks that are part of its associated transaction to the STM and then notifies transaction termination to the STM. Then, the transaction enters a termination phase local to the STM. If the transaction commits successfully, the STM is updated ; if the transaction fails, all memory modifications made within this transaction are undone, the process can be restarted from its previous memory state.

There is no fundamental difference between the operation of the tc_FTMP and that of a non-fault tolerant shared memory multiprocessor. Processes access memory blocks through caches and modified blocks are written back to the STM to make room for new

blocks when all blocks in cache are modified. A non-fault tolerant multiprocessor generally writes blocks back one at a time. In the tc_FTMP architecture, the non write through nature of the cache allows several modifications on the same memory block before it is updated into main memory. The cache write back to the STM in a *single flush* all the blocks modified by a process within its associated transaction (since the last flush of the blocks modified by this process).

When processors in a tightly coupled multiprocessor share memory, there are two related problems, a cache coherence problem and a main memory coherence problem. The cache coherence problem can be solved by a cache coherence protocol [Arch87] responsible for keeping coherent cached copies of blocks. We also need to keep process states consistent with respect to main memory to allow recovery after a crash.

Suppose that process P_2 reads a block b *owned* by process P_1 (the process that last wrote on a block is said to be the *owner* of this block; if this block has been modified before by other processes, it may not be part of the process's associated transaction). If P_1 fails and has not been checkpointed after writing into b, the STM undoes the write into b. At this point, block b has two different values if read by P_1 or P_2.

To avoid such inconsistencies, a first approach would consist of forcing termination of P_1's associated transaction at the time P_2 modifies b so that write into be can't be undone by the STM after P_2 has read b. This would considerably lower system performance when sharing is high (for example if b contains a synchronization variable, one process should be checkpointed before each access to b).

The approach chosen for the tc_FTMP architecture avoids too frequent checkpoints by having each process keep track of its *dependencies* on other processes. Two or more processes are said to be *dependent* when they have both modified the same block (similarly, transactions associated with dependent processes are also said to be dependent). To keep memory state consistent, dependent transactions are committed atomically, that is, if a cache initiates termination of a transaction, all dependent transactions must terminate. Because of this approach processes may share memory blocks in a very efficient way.

In the following sections, we review the main aspects of the tc_FTMP architecture: architecture, stable memory and cache operation and two protocols, one to manage data sharing, the other to ensure atomic update of main memory.

3.2 Architecture

The tc_FTMP architecture consists of processor elements and a shared *Stable Transactional Memory* connected by a system bus. The only shared memory in the architecture is the Stable Transactional Memory.

To speed memory access time, the tc_FTMP provides a *non write-through* cache memory for each processor containing copies of recently referenced memory blocks. The non write-through nature of the cache allows a processor to perform multiple writes on the same block before writing it back to main memory. The cache may be flushed due to an overflow or on operating system's request (the operating system flushes caches after a predefined time to reduce computation overhead after recovering a process).

For our scheme to work, we need the following assumptions :

- Caches share the identifier of the current process with their associated processor (in order to identify the process modifying a block).
- When a block is placed on the bus by a cache, the cache also supplies the identifier of the transaction associated with the process that owns this block (or a null identifier if the block is not part of a transaction).

3.3 Stable Transactional Memory

One important part of the tc_FTMP is the *Stable Transactional Memory* (STM). It is a non volatile fail-free random access memory divided into fixed size memory blocks supporting an atomic transaction mechanism. The STM is comprised of several modules connected on the shared bus of the tc_FTMP architecture. Each module consists in two memory banks associated with a snoopy controller. To give the STM the capability to undo some memory changes, each block possesses two copies in the STM; one current copy and one backup copy.

When a block which is not yet part of a transaction is modified by a process, that block is included in the transaction associated with this process (the cache detects that the block is not part of a transaction and sends a command to the STM which logs the block as part of the transaction).

Upon transaction termination, an update of the blocks has to be made atomically on both banks. This update consists in three steps :

1. The blocks modified by a process are written by the cache into the STM which copies them into first memory bank.
2. The cache sends an *end_transaction* message to the STM.
3. When all the blocks which are part of the transaction have been written back to the STM, the STM copies them onto a second memory bank.

If the cache crashes during step 1 or 2, the STM detects the failure when its associated transaction is timed out and previous memory state is recovered by the STM (blocks are copied from second bank onto first bank). The transaction aborts, all the memory modifications it has performed are undone. Once the *end_transaction* message has been received by the STM, any subsequent read will return the values written at step 1.

The STM also supports *transaction groups* to allow dependent transactions to be committed atomically. A transaction group consists of the atomic update of memory blocks modified by several processes into the STM (ie, blocks that are part of dependent transactions). The STM guarantees atomicity of these updates. Transaction groups are created by caches when they initiate the termination of dependent transactions. A transaction group terminates if all the transactions of the group terminate and aborts if at least one of the transactions aborts. If a group aborts, all its constituent transactions abort.

Transaction groups differ from nested transactions [Moss81] in that nested transactions are a mean for structuring programs and allow transaction recovery with a sub-transaction granularity. Transaction groups are only a mechanism that permits the atomic commit of dependent transactions.

Create_Transaction()

Creates a transaction. The transaction identifier is generated by the STM and returned to the calling cache. The STM initializes a timeout associated with this transaction.

End_Transaction(t)

If all the blocks which are part of transaction t have been written back to the STM, the STM copies internally blocks from bank 1 onto bank 2. If some blocks modified have not been written back, memory modifications undertaken by this transactions are canceled.

Create_Group()

Creates a transaction group.

Add_Group(t)

Adds transaction t to the transaction group. If t is already part of the group this command is ignored.

End_Group()

If all the blocks owned by processes whose transactions are part of the group have been written back to the STM, the STM internally copies blocks from bank 1 onto bank 2. If not memory modifications undertaken by all the transactions participating to the group are undone.

Add_Block(t,b)

Block b has been modified by a process and is not already part of a transaction. The STM logs that b is part of transaction t.

Write_Block(b)

Cache writes back block b.

Figure 1: STM commands

The commands recognized by the STM are summarized in figure 1 (all commands are transmitted on the bus by the caches). The STM distinguishes between process identifiers and transaction identifiers since a process may be associated with more than one transaction when it has migrated from one processor to another. The migration problem will be detailed in section 3.7.

3.4 Sharing memory

In order for multiple processors to share memory blocks, the caches in the tc_FTMP architecture support cache coherence protocol with invalidation very similar to those used in non fault tolerant shared memory multiprocessors [Arch87]. This protocol serves two purposes :

- Efficient data sharing, that is reduces access to main memory as much as possible, by inhibiting memory from supplying cached copies of blocks.
- To log dependencies between processes when they update common shared blocks.

For each block in the cache, each cache maintains the following information :

- Local state.
- Identifier of the transaction associated with the process that owns the block (*nil* if the block is not part of a transaction).

The local states of a block used by the protocol (with abbreviations given in parentheses) are :

- Invalid (I)
  Copy is not up-to-date.
- Shared (S)
  The block has not been modified since it was loaded into the cache. Copies of this block may exists in other caches.
- Modified Exclusive (M)
  The block is part of a transaction, it is modified with respect to main memory. No other copies exist. The cache is the last writer of the block and is responsible for writing the block back to main memory.
- Modified Shared (O)
  The cache is the last writer of the block and is responsible for writing the block back to main memory. At least one copy in local state S exists in another cache.

Figure 2 contains a state transition diagram for the protocol which works as follows :

- **Read Hit.**
  Read takes place immediately.
- **Read Miss.**
  - If no other cache has a copy of the block, it is supplied by the STM.
  - If there exists a copy in state M or O, the cache with that copy must supply the block directly to the requesting cache and then set its local state to O.
  - If there exists a copy in state S, the cache with that copy supplies the block directly to the requesting cache.

  The block is always loaded in state S.
- **Write Hit.**

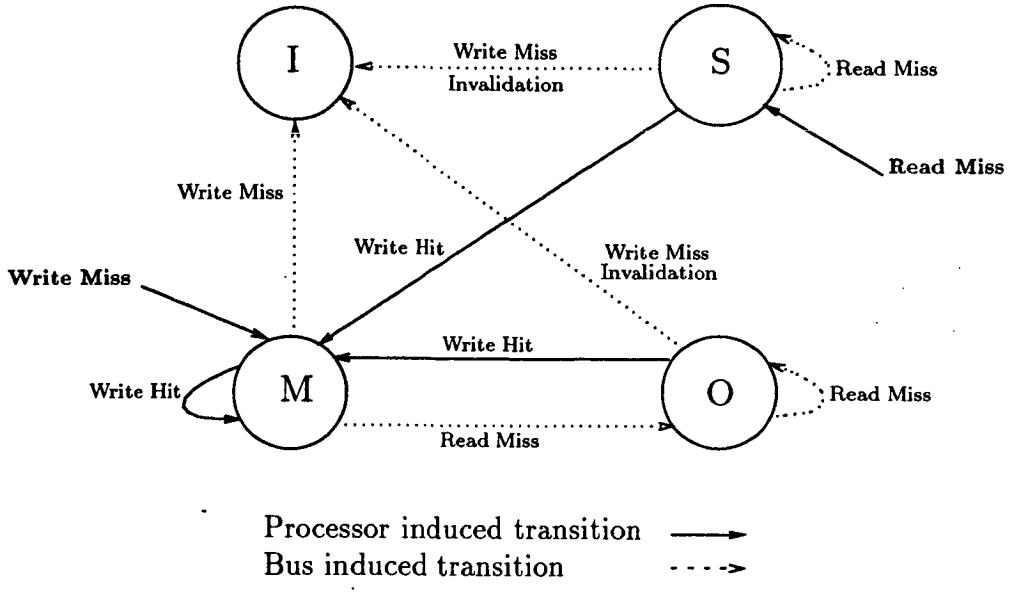

Figure 2: Coherence protocol state transition diagram

- If the block is already in state M, the write can take place immediately.
- If the block is in state O or S, an invalidation signal is sent on the bus and all other caches invalidate their copies. The local state of the block is changed to M.

- **Write Miss.**
  If there exists a copy of the block in another cache, the cache with that copy supplies the block. All caches with copies of the block invalidate those copies and the block is loaded in state M. Otherwise the STM supplies the block.

It should be noted that with this protocol, once a block is part of a transaction, that block will not be supplied by the STM to any cache until that transaction terminates. Note also that only one copy of an updated block in state M or O exists, in the cache associated with the processor that executes the process owning this block. This cache is responsible for writing the block back to the STM on transaction termination. This coherence protocol operates very much like Berkeley coherence protocol [Katz85] with the difference that in Berkeley scheme, a block always comes from memory unless a modified copy exists into a cache.

In the cases of Write Hit or Write Miss, if no transaction exists for the current process (its previous transaction has committed), the cache sends a Create_Transaction message to the STM. If the block modified is not yet part of a transaction, the cache sends an Add_Block message to the STM. The STM records that the block is part of the transaction associated with current process which becomes the owner of the block.

3.5 Keeping track of dependencies

Caches create a dependency between transaction T_P and $T_{P'}$, associated with process P and P' respectively when P accesses a block already modified by P'. A dependency is created even if the two processes reside on the same cache.

If the block to be written is already part of a transaction, the cache logs a new dependency between the transaction associated with current process and the transaction already associated with the owner of the block (whose identifier is already stored in the cache or is sent on the bus along with block content).

Dependencies are characterized into two sets for each process P :

- A set of transactions L_P associated with processes that have invalidated blocks owned by P (processes to which P has supplied blocks).
- A set of transactions I_P associated with processes owning block b when P modified b for the first time (processes that have supplied blocks to P).

For each process P, caches maintain internally :

- Its I_P set.
- An inv_P flag, set when a block owned by P is invalidated by another process (that is, when L_P is not empty).

The L_P set is maintained by a cache to keep transaction associated information as small as possible. Caches can construct L_P by broadcasting messages over the bus (if L_P is not empty, some processes have invalidated blocks owned by P and hold P's associated transaction in a $I_{P'}$ set).

All the transactions on which P's associated transaction T_P directly or transitively depends form its *dependency graph*. Figures 3 and 4 present a sample dependency graph and associated information after the following writes :

- Block b_1 is written successively by processes P_A and P_B.
- Block b_2 is written successively by processes P_B and P_A.
- Block b_3 is written successively by processes P_C, P_B and P_D.

In the graph, an arc between transaction T_i and T_j labeled b means that process j has accessed block b which was previously owned by process i. T_i is a member of I_j and T_j is a member of L_i.

All of this information is used by the caches to implement the transaction termination protocol which is described in next section.

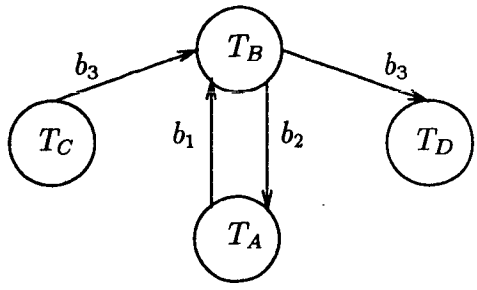

Figure 3: Sample dependency graph

Process (P_i)	Transaction (T_i)	Blocks part of associated transaction	Blocks Owned	Dependency Sets		
				L_i	I_i	Inv Flag
P_A	T_A	b_1	b_2	T_B	T_B	True
P_B	T_B	b_2	b_1	T_A,T_D	T_A,T_C	True
P_C	T_C	b_3		T_B		True
P_D	T_D		b_3		T_B	False

Figure 4: Associated informations

3.6 Updating shared memory

The transaction termination protocol behavior is dependent on whether a process's dependency sets are empty or not. If the process's dependency sets are empty, the blocks it has modified are simply written back to the STM. The cache then sends a transaction termination message to the STM. After writing a block back into the STM the cache changes the local state recorded for the block to S, so that data remains valid for future accesses.

When the process's dependency sets are not empty the cache initiating termination creates a transaction group to force atomic commit of all dependent transactions. Since caches do not maintain the entire dependency graph, building the group requires the cooperation of all caches in the system. For example, suppose that cache C_A, associated with the processor executing process A in the example from figure 3, wants to terminate transaction T_A. Since the information stored in cache reveals that T_A's dependency sets are not empty a transaction group must be created. To initiate this, cache C_A sends a *Create_Group* command to the STM.

Then, the initiating cache includes the terminating transaction into the group as well as its I set (in our example transactions T_A and T_B are added to the group).

To ensure all caches write back any block owned by processes whose transactions are member of the group, the initiating cache broadcasts a *terminate* message for each transaction member of I set (cache C_A sends a *terminate* message for T_B). Upon receiving those messages, caches repeat these actions, add transactions into the group and send other *terminate* messages (cache C_B adds transactions T_A and T_C to the group and sends

a *terminate* message for those transactions). Transactions may be added more than once into the group, since the STM does not take this in account.

At this point, a transaction that is not member of the I_P set of any process, has not been included into the group (transaction T_D in our example, is the last writer of block b_3 and is not member of any I_P set). To force those transactions into the group caches use the *inv* flag. For each transaction T that has its *inv* flag set, caches broadcast a *flush* message containing T's identifier. Upon receiving this *flush* message, the other caches check if they hold transactions depending on I and if so act as if they have received a *terminate* message for those transactions (in our example transaction T_A, T_B and T_C have their *inv* flag set. Cache C_D receives the *flush* message for T_B and places transaction T_D into the group since T_B is a member of I_D).

All caches write the blocks owned by the transactions into the STM and then send an *end* message to the initiating cache. Blocks are invalidated once they have been written back. This ensures that if the transaction group aborts, every process accessing a block written back will obtain that block from the STM. In order to send the *End_Group* command to the STM the initiating cache maintains the group internally. When it has received as many *end* messages as there are transactions in the group, it sends the *End_Group* command to the STM. The effect of this last message is to have all the transactions of the group committed.

The initiating cache performs the following algorithm :

```
Create_Group;                { create a transaction group }

Add_Group(P.transaction);
for each i from P.I_set do
  Add_Group(i);              { add I_set into the group }
  group:=group+i;
  Broadcast(<terminate,i>);
end;

if P.inv = true then   { other processes own blocks modified by P }
  Broadcast(<flush,P.transaction>);

for each block b do
  if (b.state=M or b.state=O) and b.owner=P then
    Write_Block(b);
    b.state:=I;
    if b.state=O then        { other caches may have a copy }
      Broadcast(<invalidate,b>);
  end;
end;

while cardinal(group<>1) do
  Wait(<end,T'> or <terminate,T'>);
  if <terminate,T'> then { a new process has been added into the group }
    group:=group+T';
  if <end,T'> then
   { all the blocks owned by a process have been written back }
```

```
    group:=group-T';
end;

End_Group.
```

Upon receiving a $< terminate, T' >$ message the cache associated with the processor executing process P' associated with T', performs the following actions :

```
{ T' is not added to the group, since this has already be done }
for each i from P'.I_set do
  Add_Group(i);
  Broadcast(<terminate,i>);
end;

if P'.inv = true then
Broadcast(<flush,T'>);

< write back all the blocks owned by P' >
< invalidate cached copies of blocks owned by P'>

Broadcast(<end,T'>);
```

Upon receiving a $< flush, T >$ message, each cache associated with a processor executing a process P' that has T in its I set adds P' in the transaction group and performs the following actions :

```
Add_Group(P'.transaction)
for each i from P'.I_set do
  Add_Group(i);
  Broadcast(<terminate,i>);
end;

if P'.inv = true then
  Broadcast(<flush,P'.transaction>);

< write back all the blocks owned by P' >
< invalidate cached copies of blocks owned by P'>

Broadcast(<end,P'.transaction>);
```

This termination protocol ensures that the whole dependency graph is put into the group whichever cache initiates termination. Owing to the transaction group facilities provided by the STM, dependent process states are updated atomically. By constructing the graph dynamically each cache does not need to create and maintain the entire dependency graph for each transaction. We could also avoid keeping the group into the initiating cache by sending unconditionally the *End_Group* command after a time out initiated with the maximum flush time for a transaction group.

Since caches do not maintain the entire dependency graph for each transaction, only one transaction group is allowed at a time because a cache wishing to terminate a transaction is unable to determine if it is part of an in-progress group or if it needs to create a new

group for this transaction. Although only one transaction group at a time may be active at a time, transactions that do not depend on any other transactions may be terminated at the same time as a group. For the same reason, caches may put on the bus some useless messages since, except the cache initiating group termination, other caches do not have any knowledge of transactions currently added to the group. This is the price to pay for keeping cache information relatively small.

3.7 Process migration and recovery

The tc_FTMP architecture handles process migration and process recovery in a similar way, by relocating processes from one processor to another. Since memory is kept consistent by the transaction mechanism, we can recover processes by restarting their execution on another processor. We now describe in turn migration and recovery.

To have a process P migrate from processor CP_1 to processor CP_2, the kernel running on processor CP_1 simply removes P from its process queue and places it into that of CP_2; creating a dependency between the kernels running on the two processors (usually, shared memory multiprocessors share a single process ready queue. The tc_FTMP architecture has a process queue for each processor to avoid the creation of dependencies between the kernels of different processors when they schedule processes). The operating system terminates the transactions associated with the two kernels processes to ensure that migration is effectively done even if a fault occurs. When P is activated on CP_2, a new transaction T_{P2} is created. Process P may then modify some of its blocks which are located on cache C_1, creating a dependency between T_{P2} and the transaction it already had on C_1, T_{P1}. If one of the processors fail during the update of the process queues, migration is undone by the STM.

Failure recovery is done in a similar way. When the failure of a processor is detected, the processes it was running are migrated onto other processors. One problem is that the blocks loaded into the failed cache are no longer accessible. When the transactions associated with failed processes are aborted by the STM due to a time out, the blocks become accessible again and the STM can supply them on demand. In the same way, any transaction group containing transactions associated with failed processes is aborted because the failed cache cannot write blocks back into the STM when the group terminates.

Although this architecture solves the data sharing problem set by fault tolerant tightly coupled architectures, the overhead for achieving this behavior may be too high for our approach to be viable. According to the simulation results presented in next section, this is not the case.

4 Performance Evaluation

This section we discusses the results of simulations which were conducted to evaluate the performance of the tc_FTMP architecture.

The simulation model is based on one proposed by [Arch87]. This model was chosen because it allows the simulation of various architecture and cache coherence protocols and

accurately reflect the differences between them. We use trace driven simulation with a synthetic workload model which allows us to easily vary the level of data sharing. Our simulator emulates the tc_FTMP architecture. The operation of processors, caches and bus is similar to those described in [Arch87], but incorporating the above coherence and transaction termination protocols. The metric used in determining system performance is the sum of all processor utilizations in the system. Processor utilization is measured by the ratio of the total inter-reference time to the simulation time.

We make the following assumptions about the operation of the STM :

1. The STM is not distributed (only one memory element on the bus).

2. Read access to the STM is 1.5 times slower than for an typical RAM memory; write access is 2.5 times slower. These figures are consistent with published data [Bana88].

In order to evaluate overhead of the protocols needed for fault tolerance we compare our architecture with a non fault tolerant multiprocessor running the same memory coherence protocol. The differences between the two simulations are :

- Caches write blocks back to shared memory one at a time instead of flushing groups of blocks.

- There is no penalty for shared main memory access.

- Caches do not generate the bus messages required by transaction management.

Figure 5 presents the simulation results for the two protocols for one to sixteen processors with a high sharing level. In this simulation, fault tolerance decreases overall performance by about 25 percent due to the penalty in accessing the STM and managing the transaction protocols within the caches.

The a second simulation results demonstrate a different approach in which no data sharing is allowed. In this scheme, once a block has been modified, any access to this block made by another processor results in forcing termination of the transaction associated with the owner of the block. Figure 6 presents the results of this simulation. Obviously, such a scheme reduces performances although it simplifies transaction management.

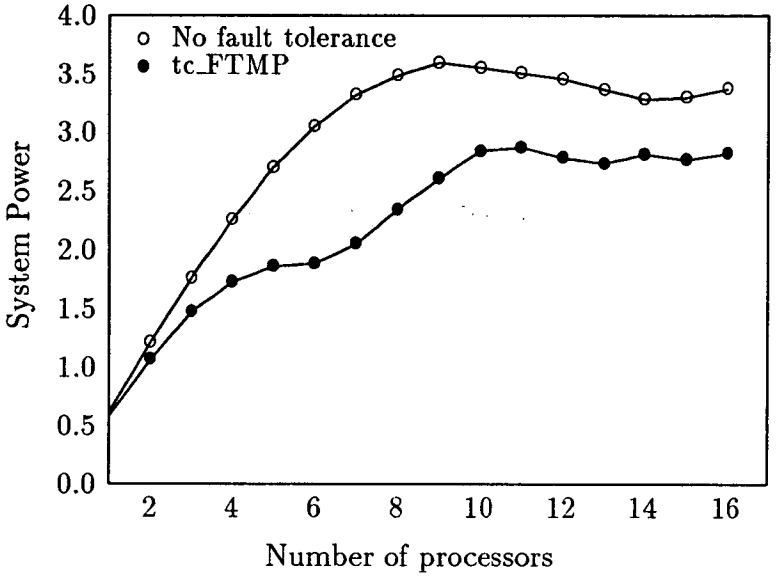

Figure 5: Overhead for fault tolerance

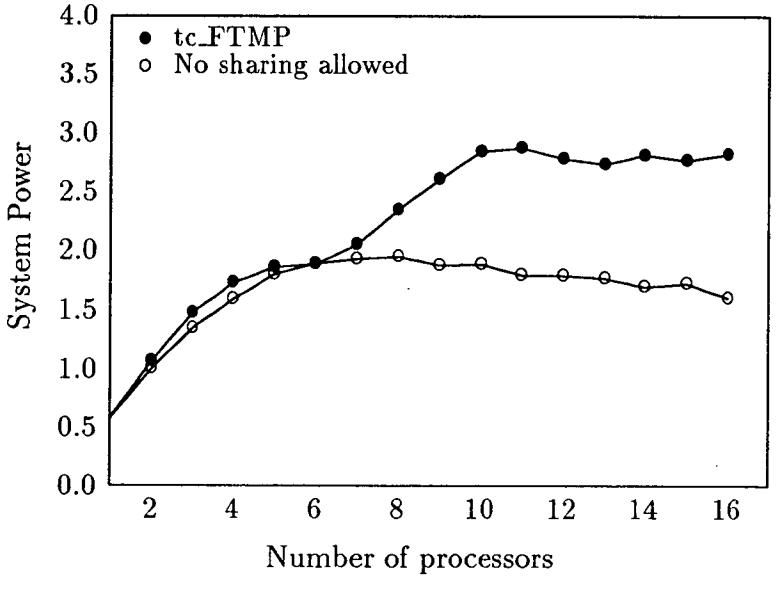

Figure 6: Influence of data sharing protocol

5 Summary and conclusion

This report has reviewed the main aspects of the tc_FTMP architecture which attains fault tolerance by using a stable shared memory to maintain processes states recoverable from processor and cache failures, Simulations of its potential performance have also been presented. The main originality of this approach is the way it allows write sharing of data blocks in the same way as in non fault tolerant shared memory multiprocessors while keeping dependent processes states coherent.

Though the STM functionalities are fully defined, we are still studying its physical implementation, taking advantage of our previous experiences in the building of stable storage which operates very much like the STM. Much work has still to be done, especially on cache coherence and group termination protocols which are still opened to improvement, but we believe that the cost of fault tolerance in such an architecture should be relatively low, according to simulation results and with respect to the assumptions made for the simulation model.

Acknowledgments

The authors wish to thank J. P. Banâtre, P. A. Lee, C. Morin and B. Rochat for their helpful comments on earlier drafts.

References

[Arch87] J. K. Archibald. The Cache Coherence Problem in Shared-Memory Multiprocessors. Technical Report, University of Washington, 1987.

[Bana86] J. P. Banâtre, M. Banâtre, G. Lapalme and Fl. Ployette. The Design and Building of ENCHERE, a Distributed Electronic Marketing System. *Communications of the ACM*, 29(1):19–29, 1986.

[Bana88] J. P. Banâtre, M. Banâtre, and G. Muller. Ensuring Data Security and Integrity with a Fast Stable Storage. In *Proceedings of the 4th International Conference on Data Engineering*, pages 285–293, February 1988.

[Bart87] J. Bartlett, J. Gray, and B. Horst. Fault Tolerance in Tandem Computer Systems. In *Dependable Computing and Fault Tolerant Systems*, Springer Verlag, 1987.

[Bern88] P. A. Bernstein. Sequoia: a Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing. *Computer*, 37–45, February 1988.

[Borg89] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[Harr87] E. S. Harrison and E. Schmitt. The Structure of System/88, a Fault-Tolerant Computer. *IBM Systems Journal*, 26(3):293–318, 1987.

[Hopk78] A. Hopkins. FTMP - A Highly Reliable Fault Tolerant Computing System. *Proceedings of the IEEE*, 66(10):1221–1239, October 1978.

[Katz85] R. Katz, S. Eggers, D.A. Wood, C. Perkins, and R.G. Sheldon. Implementing a Cache Consistency Protocol. In *Proceedings of 12th International Symposium on Computer Architecture*, pages 276–283, 1985.

[Lamp76] B.W. Lampson and H. Sturgis. Crash Recovery in a Distributed Data Storage System. *Working Paper, Xerox Palo Alto Research Center*, November 1976.

[Lamp81] B. Lampson Atomic Transactions. in *Distributed Systems and Architecture and Implementation : an advanced course*, LNCS 105 Springer Verlag, 1981.

[Moss81] J. E. B. Moss Nested Transactions : An Approach to Reliable Computing. MIT Laboratory for Computer Science Technical Report 260, April 81 also available in book from MIT Press

[Rand78] B. Randell, P.A. Lee and P.C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys*, 10(2):123-165, June 1978.

[Smit82] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

LISTE DES DERNIERES PUBLICATIONS INTERNES

PI 506 **VM pRAY, AN EFFICIENT RAY TRACING ALGORITHM ON DISTRIBUTED MEMORY PARALLEL COMPUTER**
Didier BADOUEL, Thierry PRIOL
Janvier 1990, 50 Pages.

PI 507 **ON THE REGULAR STRUCTURE OF PREFIX REWRITINGS**
Didier CAUCAL
Janvier 1990, 32 Pages.

PI 508 **RAY TRACING ON DISTRIBUTED MEMORY PARALLEL COMPUTERS: STRATEGIES FOR DISTRIBUTING COMPUTATIONS AND DATA.**
Didier BADOUEL, Kadi BOUATOUCH, Thierry PRIOL
Janvier 1990, 16 Pages.

PI 509 **STABILITY ANALYSIS AND IMPROVEMENT OF THE BLOCK GRAM-SCHMIDT ALGORITHM**
William JALBY, Bernard PHILIPPE
Janvier 1990, 24 Pages.

PI 510 **TESTING FOR THE UNBOUNDEDNESS OF FIFO CHANNELS IN PROGRAMS.**
Thierry JERON
Janvier 1990, 30 Pages.

PI 511 **AUTOMATIC ANIMATION CONTROL OF PHYSICAL SYSTEMS.**
Georges DUMONT, Bruno ARNALDI, Gérard HEGRON
Janvier 1990, 22 Pages.

PI 512 **A FAULT TOLERANT TIGHTLY COUPLED MULTIPROCESSOR ARCHITECTURE BASED ON STABLE TRANSACTIONAL MEMORY**
Michel BANATRE, Philippe JOUBERT
Février 1990, 20 Pages.

PI 513 **BUILDING A GLOBAL TIME ON PARALLEL MACHINES**
Jean-Marc JEZEQUEL
Février 1990, 28 Pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique